Introduction to Operating Systems

What's an OS?

The OS is a layer between applications and hardware to ease development.

- **Abstraction**. provides abstraction for applications:
  - manages + hides hardware details
  - uses low-level interfaces (not available to applications)
  - multiplexes hardware to multiple programs (*virtualization*)
  - makes hardware use efficient for applications
- **Protection**.
  - from processes using up all resources (*accounting*, *allocation*)
  - from processes writing into other processes memory
- **Resource Management**.
  - manages + multiplexes hardware resources
  - decides between conflicting requests for resource use
  - *goal*: efficient + fair resource use
- **Control**.
  - controls program execution
  - prevents errors and improper computer use

⇝ **no universally accepted definition**

Hardware Overview

- **Bus**: CPU(s)/devices/memory (conceptually) connected to common bus
  - CPU(s)/devices competing for memory cycles/bus
  - all entities run concurrently
  - *today*: multiple buses
- **Device controller**: has local buffer and is in charge of particular device
- **Interplay**:
  1. CPU issues commands, moves data to devices
  2. Device controller informs APIC that it has finished operation
  3. APIC signals CPU
  4. CPU receives device/interrupt number from APIC, executes handler

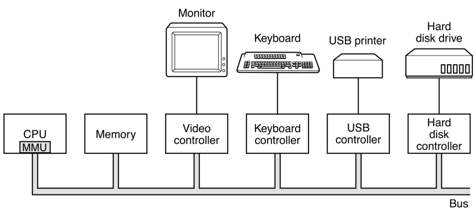


Figure 1: Traditional bus design.

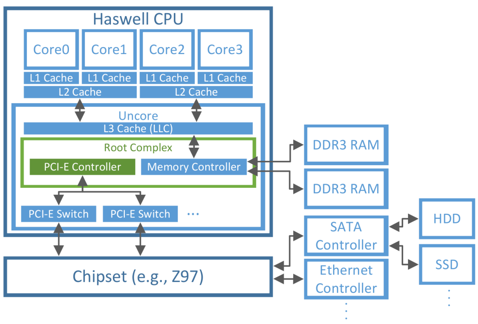


Figure 2: Modern bus design.

Central Processing Unit (CPU) — Operation

- **Principle**:
  1. *fetches* instructions from memory,
  2. *executes* them
- **During execution**: (meta-)data is stored in CPU-internal registers, i.e.
  - general purpose registers
  - floating point registers
  - instruction pointer (IP)
  - stack pointer (SP)
  - program status word (PSW)

CPU — Modes of Execution

- **User mode** (x86: *Ring 3*/*CPL 3*):
  - only non-privileged instructions may be executed
  - cannot manage hardware → *protection*
- **Kernel mode** (x86: *Ring 0*/*CPL 0*):
  - all instructions allowed
  - can manage hardware with *privileged instructions*

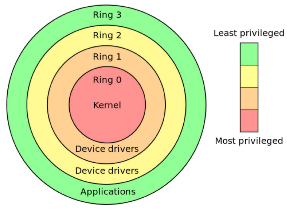


Figure 3: The different protection layers in the *ring model*.

Random Access Memory (RAM)

- **Principle**: keeps currently executed instructions + data
- **Connectivity**:
  - *today*: CPUs have built-in **memory controller**
  - *CPU caches*: "wired" to CPU
  - *RAM*: connected via pins
  - *PCI-E switches*: connected via pins

Caches

- **Problem**: RAM delivers instructions/data slower than CPU can execute
- **Locality principle**:
  - *spatial locality*: future refs often near previous accesses (e.g. next byte in array)
  - *temporal locality*: future refs often at previously accessed ref (e.g. loop counter)
- **Solution**: *caching* helps mitigating this memory wall
  1. *copy* used information temporarily from slower to faster storage
  2. *check* faster storage first before going down *memory hierarchy*
  3. if *not found*, data is copied to cache and used from there
- **Access latency**:
  - *register*: ~1 CPU cycle
  - *L1 cache* (per core): ~4 CPU cycles
  - *L2 cache* (per core pair): ~12 CPU cycles
  - *L3 cache/LLC* (per uncore): ~28 CPU cycles (~25 GiB/s)
  - *DDR3-12800U RAM*: ~28 CPU cycles + ~ 50ns (~12 GiB/s)

Device controlling

- **Device controller**: controls device, accepts commands from OS via *device driver*
- **Device registers/memory**:
  - *control* device by writing device registers
  - *read* status of device by reading device registers
  - *pass data* to device by reading/writing device memory
- **Device registers/memory access**:
  1. **port-mapped IO** (PMIO): use special CPU instructions to access port-mapped registers/memory
  2. **memory-mapped IO** (MMIO):
     - use same address space for RAM and device memory
     - some addresses map to RAM, others to different devices
     - access device's memory region to access device registers/memory
  3. **Hybrid**: some devices use hybrid approaches using both

Summary

- The OS is an **abstraction** layer between applications and hardware (multiplexes hardware, hides hardware details, provides protection between processes/users)
- The CPU provides a **separation** of User and Kernel mode (which are required for an OS to provide protection between applications)
- CPU can execute commands faster than memory can deliver instructions/data — memory **hierarchy** mitigates this memory wall, needs to be carefully managed by OS to minimize slowdowns
- device drivers **control** hardware devices through PMIO/MMIO
- Devices can **signal** the CPU (and through the CPU notify the OS) through interrupts

OS Concepts

OS Invocation

- OS Kernel does **not** always run in background!
- Occasions invoking kernel, switching to kernel mode:
  1. **System calls**: User-Mode processes require higher privileges
  2. **Interrupts**: CPU-external device sends signal
  3. **Exceptions**: CPU signals unexpected condition

System Calls — Motivation

- **Problem**: protect processes from one another
- **Idea**: Restrict processes by running them in user-mode
- ⇝ **Problem**: now processes cannot manage hardware,...
  - who can switch between processes?
  - who decides if process may open certain file?
- ⇝ **Idea**: OS provides **services** to apps
  1. app calls system if service is needed (**syscall**)
  2. OS checks if app is allowed to perform action
  3. if app may perform action and hasn't exceeded quota, OS performs action in behalf of app in kernel mode

System Calls — Examples

- `fd = open(file, how,...)` – open file for read/write/both
- documented e.g. in `man 2 write`
- overview in `man 2 syscalls`

System Calls vs. APIs

- **Syscalls**: interface between apps and OS services, limited number of well-defined entry points to kernel
- **APIs**: often used by programmers to make syscalls (e.g. `printf` library call uses `write` syscall)
- common APIs: Win32, POSIX, C API

System Calls — Implementation

- **Trap Instruction**: single syscall interface (entry point) to kernel
  - switches CPU to kernel mode, enters kernel in same way for all syscalls
  - *system call dispatcher* in kernel then acts as syscall multiplexer
- **Syscall Identification**: number passed to trap instruction
  - *Syscall Table* maps syscall numbers to kernel functions
  - *Dispatcher* decides where to jump based on number and table
  - programs (e.g. `stdlib`) have syscall number compiled in!

  ⇝ never reuse old syscall numbers in future kernel versions

Interrupts

- **Devices**: use interrupts to signal predefined conditions to OS
  - *reminder*: device has "interrupt line" to CPU (e.g. device controller informs CPU that operation is finished)
- **Programmable Interrupt Controller**: manages interrupts
  - interrupts can be *masked* (queued, delivered when interrupt unmasked)
  - queue has finite length ⇝ interrupts can get lost
- **Examples**:
  1. *timer-interrupt*: periodically interrupts processes, switches to kernel ⇝ can then switch to different processes for fairness
  2. *network interface card* interrupts CPU when packet was received ⇝ can deliver packet to process and free NIC buffer
- **Interrupt process**:
  1. CPU looks up *interrupt vector* (= table pinned in memory, contains addresses of all service routines)
  2. CPU transfers control to respective *interrupt service routine* in OS that handles interrupt

  ⇝ interrupt service routine must first save interrupted process's state (instruction pointer, stack pointer, status word)

Exceptions

- **Motivation**: unusual condition → impossible for CPU to continue processing
- ⇝ **Exception** generated within CPU:
  1. CPU interrupts program, gives kernel control
  2. kernel determines reason for exception
  3. if kernel can resolve problem ⇝ does so, continues *faulting instruction*
  4. kills process if not
- **Difference to Interrupts**: interrupts can happen in any context, exceptions always occur asynchronous and in process context

OS Concepts — Physical Memory

- up to early 60s:
  - programs loaded and run directly in *physical memory*
  - program too large → partitioned manually into *overlays*
  - OS: swaps overlays between disk and memory
  - different jobs could observe/modify each other

OS Concepts — Address Spaces

- **Motivation**: bad programs/people need to be isolated
- **Idea**: give every job the illusion of having all memory to itself
  - every job has own *address space*, can't name addresses of others
  - jobs always and only use virtual addresses

Virtual Memory — Indirect Addressing

- **MMU**: every CPU has built-in *memory management unit* (MMU)
- **Principle**: translates virtual addresses to physical addresses at every load/store

  ⇝ address translation protects one program from another
- **Definitions**:
  - *Virtual address*: address in process' address space
  - *Physical address*: address of real memory

Virtual Memory — Memory Protection

- **Kernel-only Virtual Addresses**
  - kernel typically part of all address spaces
  - ensures that apps can't touch kernel memory
- **Read-only virtual addresses**: can be enforced by MMU
  - allows safe sharing of memory between apps
- **Execute Disable**: can be enforced by MMU
  - makes code injection attacks harder

Virtual Memory — Page Faults

- **Motivation**: not all addresses need to be mapped at all times
  - MMU issues *page fault* exception when accessed virtual address isn't mapped
  - OS handles page faults by loading faulting addresses and then continuing the program

  ⇝ memory can be *over-committed*: more memory than physically available can be allocated to application
- **Illegal addresses**: page faults also issued by MMU on illegal memory accesses

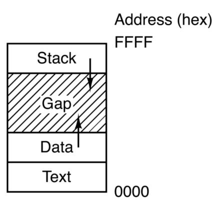
OS Concepts — Processes

- **Process**: program in execution ("instance" of program)
- each process is associated with
  - **Process Control Block** (PCB): contains information about allocated resources
  - virtual **Address Space** (AS):
    - all (virtual) memory locations a program can name
    - starts at 0 and runs up to a maximum
    - address 123 in AS1 generally ≠ address 123 in AS2
    - indirect addressing ⇝ different ASes to different programs

    ⇝ *protection between processes*

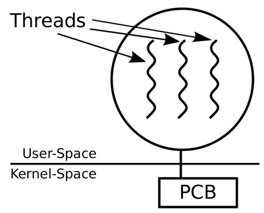
OS Concepts — Address Space Layout

- **Sections**: address spaces typically laid-out in different sections
  - memory addresses between sections *illegal*
  - illegal addresses ⇝ page fault (*segmentation fault*)
  - OS usually kills process causing segmentation fault
- **Important sections**:
  - *Stack*: function history, local variables
  - *Data*: Constants, static/global variables, strings
  - *Text*: Program code



OS Concepts — Threads

- **Thread**: represents execution state of process (≥ 1 thread per process)
  - *IP*: stores currently executed instruction (address in `text` section)
  - *SP*: stores address of stack top (> 1 threads → multiple stacks!)
  - *PSW*: contains flags about execution history (e.g. last calculation was 0 → used in following jump instruction)
  - more general purpose registers, floating point registers,...



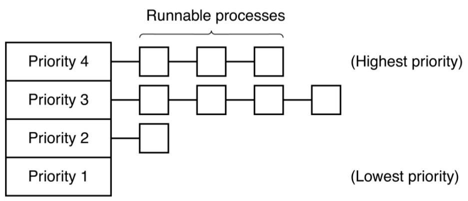
OS Concepts — Policies vs. Mechanisms

- **Mechanism**: implementation of what is done (e.g. commands to write to HDD)
- **Policy**: rules which decide when what is done and how much (e.g. how often, how many resources are used,...)

→ *mechanisms can be reused even when policy changes*

OS Concepts — Scheduling

- **Motivation**: multiple processes/threads available ⇝ OS needs to switch between them (for multitasking)
- **Scheduler**: decides which job to run next (*policy*) — tries to
  - provide fairness
  - meet performance goals
  - adhere to priorities
- **Dispatcher**: performs task-switching (*mechanism*)



OS Concepts — Files

- **Motivation**: OS hides peculiarities of file storage, programmer uses device-independent *files/directories*
- **Files**: associate *file name* and *offset* with bytes
- **Directories**: associate *directory names* with directory names or file names
- **File System**: ordered block collection
  - main task: translate (dir name + file name + offset) to block
  - programmer uses file system operations to operate on files (`open`, `read`, `seek`)
  - processes can communicate directly through special *named pipe* file (used with same operations as any other file)

OS Concepts — Directory Tree

- **Directories**: form *directory tree/file hierarchy* → structure data
- **Root Directory**: topmost directory in tree
- **Path Name**: used to specify file

OS Concepts — Mounting

- **Unix**: common to orchestrate multiple file systems in single file hierarchy
- file systems can be *mounted* on directory
- **Win**: manage multiple directory hierarchies with drive letters (e.g. `C:\Users`)

OS Concepts — Storage Management

- **OS**: provides uniform view of information storage to file systems
  - *Drivers*: hide specific hardware devices → hides device peculiarities
  - general interface abstracts physical properties to logical units → block
- **Performance**: OS increases I/O performance:
  - *Buffering*: Store data temporarily while transferred
  - *Caching*: Store data parts in faster storage
  - *Spooling*: Overlap one job's output with other job's input

Summary

- **OS**: provides abstractions for and protection between applications
- **Kernel**: does not always run — certain events invoke kernel
  - *syscall*: process asks kernel for service
  - *interrupt*: device sends signal that OS has to handle
  - *exception*: CPU encounters unusual situation
- **Processes**: encapsulate resources needed to run program in OS
  - *threads*: represent different execution states of process
  - *address space*: all memory process can name
  - *resources*: allocated resources, e.g., open files
- **Scheduler** decides which process to run next when multi-tasking
- **Virtual Memory** implements address spaces, provides protection between processes
- **File system** abstracts background store using I/O drivers, provides simple interface (files + directories)

Processes

The Process Abstraction

- **Motivation**: computers (seem to) do "several things at the same time" (quick process switching → *multiprogramming*)
- **Model**: *process abstraction* models this concurrency:
  - container contains information about program execution
  - conceptually, every progress has own "‘virtual CPU"’
  - execution context is changed on process switch
  - dispatcher switches context when switching processes
  - **context switch**: dispatcher saves current registers/memory mappings, restores those of next process

Process-Cooking Analogy

- Program/Process like Recipe/Cooking
- **Recipe**: lists ingredients, gives algorithm what to do when
  ⇝ program describes memory layout/CPU instructions
- **Cooking**: activity of using the recipe
  ⇝ process is activity of executing a program
- multiple similar recipes for same dish
  ⇝ multiple programs may solve same problem
- recipe can be cooked in different kitchens at the same time
  ⇝ program can be run on different CPUs at the same time (as different processes)
- multiple people can cook one recipe
  ⇝ one process can have several worker threads

Concurrency vs. Parallelism

- OS uses currency + parallelism to implement multiprogramming
  1. **Concurrency**: multiple processes, one CPU
     ⇝ not at the same time
  2. **Parallelism**: multiple processes, multiple CPU
     ⇝ at the same time

Virtual Memory Abstraction — Address Spaces

- every process has own *virtual addresses* (`vaddr`)
- MMU relocates each load/store to *physical memory* (`pmem`)
- processes never see physical memory, can't access it directly

\+ MMU can enforce protection (mappings in kernel mode)

\+ programs can see more memory than available
  - 80:20 rule: 80% of process memory idle, 20% active
  - can keep working set in RAM, rest on disk

\- need special MMU hardware

Address Space (Process View)

- **Motivation**: code/data/state need to be organized within process
  ⇝ *address space layout*
- **Data types**:
  1. *fixed size* data items
  2. data naturally *freed in reverse allocation order*
  3. data *allocated/freed "‘randomly"’*
- compiler/architecture determine how large int is and what instructions are used in text section (`code`)

- **Loader** determines based on exe file how executed program is placed in memory

Segments — Fixed-Size Data + Code

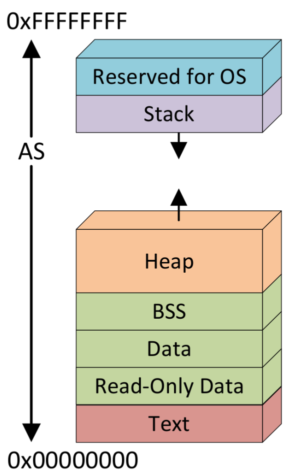
- some data in programs never changes or will be written but never grows/shrinks
  ⇝ memory can be statically allocated on process creation
- **BSS segment** (*block started by symbol*):
  - statically allocated variables/non-initialized variables
  - executable file typically contains starting address + size of BSS
  - entire segment initially 0
- **Data segment**: fixed-size, initialized data elements (e.g. global variables)
- **Read-only data segment**: constant numbers, strings
- All three sometimes summarized as one segment
- compiler and OS decide ultimately where to place which data/how many segments exist

Segments — Stack

- some data naturally freed in reverse allocation order
  - very easy memory management (stack grows upwards)
- fixed segment starting point
- store top of latest allocation in **stack pointer** (SP) (initialized to starting point)
- *allocate* `a` byte data structure: `SP += a; return(SP - a)`
- *free* `a` byte data structure: `SP -= a`

Segments — Heap (Dynamic Memory Allocation)

- some data "randomly"' allocated/freed
- two-tier memory allocation:
  1. allocate large memory chunk (**heap segment**) from OS
     - base address + **break pointer** (BRK)
     - process can get more/give back memory from/to OS
  2. dynamically partition chunk into smaller allocations
     - `malloc`/`free` can be used in random oder
     - purely user-space, no need to contact kernel



Summary

Processes: recipe vs. cooking = program vs. process

- processes = resource container for OS
- process feels alone (has own CPU and memory)
- OS implements multiprogramming through rapid process switching